SMART CONTRACT AUDIT REPORT

for

MARLIN LABS

Prepared By: Shuxiao Wang

Hangzhou, China
September 10, 2020

Document Properties

Client	Marlin Labs
Title	Smart Contract Audit Report
Target	Marlin Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Jeff Liu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	September 10, 2020	Xuxian Jiang	Final Release
1.0-rc1	September 9, 2020	Xuxian Jiang	Release Candidate #1
0.2	September 8, 2020	Xuxian Jiang	Additional Findings
0.1	September 5, 2020	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Marlin Protocol** design document and related smart contract source code, we in the report outline our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Marlin Protocol

Marlin Protocol is designed as an open protocol that provides a high-performance programmable network infrastructure for blockchain applications such as DeFi and Web 3.0. It attempts to deliver scalability, resiliency, and decentralization at layer 0 by optimizing the networking architecture underneath blockchains. On the networking layer, Marlin redesigned communication in decentralized networks by introducing a unique economic incentivization model, which allows nodes to communicate faster, thus increasing transaction throughput. Another point worth mentioning is that the protocol is compatible with all blockchains, irrespective of whether they follow Proof of Work (PoW), Proof of Stake (PoS), or any other consensus mechanisms.

The basic information of Marlin Protocol is as follows:

Table 1.1: Basic Information of Marlin Protocol

Item	Description
Issuer	Marlin Labs
Website	https://www.marlin.pro/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 10, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/marlinprotocol/Contracts/tree/phase1 (823a6cf)

1.2 About PeckShield

PeckShield Inc. [16] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the Marlin Protocol implementation. During the first phase of our audit, we studied the smart contract source code and ran our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	4	■■■■
Medium	2	■■
Low	2	■■
Informational	1	■
Total	9	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 4 high-severity vulnerabilities, 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Marlin Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Manipulatable totalRelayers	Business Logics	Fixed
PVE-002	Medium	Improved Sanity Checks of Cluster States And Their Transitions	Business Logics	Fixed
PVE-003	High	Business Logic Error in Receiver::subscribe()	Business Logics	Fixed
PVE-004	High	Business Logic Error in verifyClaim()	Business Logics	Fixed
PVE-005	Low	Improved Sanity Checks of Functions Arguments	Security Features	Fixed
PVE-006	High	Possible Pot Overdraw From FundManager	Business Logics	Fixed
PVE-007	High	Back-Running of draw() For Denial-of-Service	Business Logics	Fixed
PVE-008	Medium	Incompatibility With Deflationary ERC20 Tokens	Business Logics	Confirmed
PVE-009	Informational	Suggested Constants For Unchanged Variables	Coding Practices	Fixed

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Manipulatable totalRelayers

- ID: PVE-001
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Cluster`
- Category: Business Logics [9]
- CWE subcategory: CWE-837 [5]

Description

The Marlin Protocol organizes participating relayers into a cluster that needs to be registered in `clusterRegistry`. The cluster maintains the status of each relayer (in the `relayers` map) and the number of total participating relayers (in `totalRelayers`).

In the following, we show the code snippets of two routines, i.e., `joinCluster()` and `exitCluster()`. These two routines are used to join or leave the cluster. Our analysis shows that an already joined relayer can join again with the result of incorrectly increasing the `totalRelayers` by 1. Also, an already left relayer can leave again by further incorrectly descreasing the `totalRelayers` by 1. As a result, `totalRelayers` cannot be used to reliably keep track of the total number of current relayers.

```
    function joinCluster() public {
        relayers[msg.sender] = true;
        totalRelayers++;
    }

    function exitCluster() public {
        relayers[msg.sender] = false;
        totalRelayers--;
    }
```

Listing 3.1: Cluster.sol

Recommendation Ensure an already joined relayer does not need to join again. Similarly, an already left relayer does not need to leave again.

```
    function joinCluster() public {
        if (relayers[msg.sender] == true) { return };
        relayers[msg.sender] = true;
        totalRelayers++;
    }

    function exitCluster() public {
        if (relayers[msg.sender] == false) { return };
        relayers[msg.sender] = false;
        totalRelayers--;
    }
```

Listing 3.2: Cluster.sol

Status The issue has been fixed by this commit: `9ee8f5a692e666160b4305d152ff7223859adfa4`.

3.2 Improved Sanity Checks of Cluster States And Their Transitions

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `ClusterRegistry`
- Category: Business Logics [9]
- CWE subcategory: CWE-837 [5]

Description

The `clusterRegistry` contract defines a standard work-flow to join and/or exist a cluster. In current design, a cluster may fall in four different states, i.e., `DOESNT_EXIST`, `WAITING_TO_JOIN`, `ACTIVE`, and `EXITING`. The `DOESNT_EXIST` state means that the cluster does not exist yet; the `WAITING_TO_JOIN` state indicates that the cluster intends to be registered; the `ACTIVE` state shows the cluster is currently active; and the `EXITING` state initiates an exit process for the cluster.

```
    function addCluster(uint256 _stakeValue) public returns (bool) {
        ClusterData memory cluster = clusters[msg.sender]; //if this is updated, better
            to change it to storage
        require(
            isClustersAccepted,
            "ClusterRegistry: Clusters are not accepted"
        );
        require(
            cluster.stake.add(_stakeValue) >= minStakeAmount,
            "ClusterRegistry: Stake less than min reqd stake"
        );
        clusters[msg.sender].stake = cluster.stake.add(_stakeValue);
        if (cluster.status == ClusterStatus.DOESNT_EXIST) {
```

```
            clusters[msg.sender].status = ClusterStatus.WAITING_TO_JOIN;
            // cluster.status = ClusterStatus.WAITING_TO_JOIN;
            clusters[msg.sender].startEpoch = pot.getEpoch(block.number).add(1);
            emit ClusterJoined(msg.sender, cluster.stake.add(_stakeValue));
        } else {
            emit ClusterStakeUpdated(
                msg.sender,
                cluster.stake.add(_stakeValue)
            );
        }
        require(
            LINProxy.transferFrom(msg.sender, address(this), _stakeValue),
            "ClusterRegistry: Stake not received"
        );
        return true;
    }

    function proposeExit() public {
        if (clusters[msg.sender].exitEpoch == 0) {
            clusters[msg.sender].status = ClusterStatus.EXITING;
            uint256 exitEpoch = pot.getEpoch(block.number).add(
                clusterExitWaitEpochs
            );
            clusters[msg.sender].exitEpoch = exitEpoch;
            emit ClusterExitProposed(msg.sender, exitEpoch);
        }
    }
```

Listing 3.3: ClusterRegistry.sol

The state machine transition logic among these four states seems problematic in lacking validity checks of given cluster. In particular, we notice that a cluster can initiate the exit process even when it is not registered yet. As a result, any cluster can first schedule ahead of their exit process and then register it. By doing so, a cluster can enter and exit the registry without ever being in the `ACTIVE` state.

Recommendation Ensure that every state transition always performs sanity checks of related clusters.

Status The issue has been fixed by this commit: `9ee8f5a692e666160b4305d152ff7223859adfa4`.

3.3 Business Logic Error in Receiver::subscribe()

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Receiver`
- Category: Business Logics [9]
- CWE subcategory: CWE-841 [6]

Description

There are three types of actors in Marlin Protocol: `producers`, `relayers`, and `receivers`. The `producers` are basically the miners of a blockchain; the `relayers`, as the name indicates, relay blocks from `producers` to `receivers`; and the `receivers` can subscribe to receive block updates.

In this section, we examine the subscription logic in the `Receiver` contract. For elaboration, we show the related code snippet below. We notice a contradictory requirement in the subscription execution logic. In particular, when a new receiver intends to `subscribe()`, the protocol should not be able to find the receiver's record. As such, the requirement in lines $48-51$ will always fail, i.e., `require(receivers[i][msg.sender] > 0, "Receiver: Already subscribed to epoch")`.

```
37    // Note: Both the startEpoch and  EndEpoch are included
38    function subscribe(uint256 _startEpoch, uint256 _endEpoch) external {
39        uint256 feeToPay = subscriptionFee.mul(
40            _endEpoch.sub(_startEpoch).add(1)
41        );
42        require(_startEpoch <= _endEpoch, "Receiver: Invalid inputs");
43        require(
44            _startEpoch > pot.getEpoch(block.number),
45            "Receiver: Can't subscribe to past or current epochs"
46        );
47        for (uint256 i = _startEpoch; i <= _endEpoch; i++) {
48            require(
49                receivers[i][msg.sender] > 0,
50                "Receiver: Already subscribed to epoch"
51            );
52            receivers[i][msg.sender] = subscriptionFee;
53        }
54        require(
55            LINToken.transferFrom(msg.sender, address(this), feeToPay),
56            "Receiver: Fee not received"
57        );
58    }
```

Listing 3.4: Receiver.sol

This is certainly confusing. It turns out the requirement needs to revised as: `require(receivers[i][msg.sender] == 0, "Receiver: Already subscribed to epoch")`.

Recommendation Revise the `subscribe()` logic of adding new `receivers` as shown in the following:

```
    // Note: Both the startEpoch and  EndEpoch are included
    function subscribe(uint256 _startEpoch, uint256 _endEpoch) external {
        uint256 feeToPay = subscriptionFee.mul(
            _endEpoch.sub(_startEpoch).add(1)
        );
        require(_startEpoch <= _endEpoch, "Receiver: Invalid inputs");
        require(
            _startEpoch > pot.getEpoch(block.number),
```

```
            "Receiver: Can't subscribe to past or current epochs"
        );
        for (uint256 i = _startEpoch; i <= _endEpoch; i++) {
            require(
                receivers[i][msg.sender] == 0,
                "Receiver: Already subscribed to epoch"
            );
            receivers[i][msg.sender] = subscriptionFee;
        }
        require(
            LINToken.transferFrom(msg.sender, address(this), feeToPay),
            "Receiver: Fee not received"
        );
    }
```

Listing 3.5: Receiver.sol

Status The issue has been fixed by this commit: `9ee8f5a692e666160b4305d152ff7223859adfa4`.

3.4 Business Logic Error in verifyClaim()

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Verifier_Receiver`, `Verifier_Producer`
- Category: Business Logics [9]
- CWE subcategory: CWE-841 [6]

Description

The Marlin Protocol has a unique reward mechanism that has been developed to reward contributions to the Marlin Protocol. In particular, a portion of `LIN` is allocated as funds that are used to reward various actors in the network and the distribution is decided according to the governance.

Once awarded, various actors can submit claims to collect these rewards. During our analysis of the reward-claiming logic, we notice similar issue as reported in Section 3.3. For elaboration, we show the `verifyClaim()` logic on the `producers`. If we pay attention to the line 63, there is a requirement, i.e., `require(!rewardedBlocks[blockHash], "Block header already rewarded")`. Interestingly, this requirement follows the statement (line 62): `rewardedBlocks[blockHash] = true`. As a result, the `verifyClaims()` routine will always fail!

```
47    function verifyClaim(
48        bytes memory _blockHeader,
49        bytes memory _relayerSig,
50        bytes memory _producerSig,
```

```
        address _cluster,
        bool _isAggregated
    )
        public
        returns (
            bytes32,
            address,
            uint256
        )
    {
        bytes32 blockHash = keccak256(_blockHeader);
        rewardedBlocks[blockHash] = true;
        require(!rewardedBlocks[blockHash], "Block header already rewarded");
        bytes memory coinBase = extractCoinBase(_blockHeader);
        uint256 blockNumber = extractBlockNumber(_blockHeader);
        address actualProducer = producerRegistry.getProducer(coinBase);
        address relayer = recoverSigner(blockHash, _relayerSig);

        ...
    }
```

Listing 3.6: Verifier_Producer.sol

Apparently, a fix is to switch the orders of these two lines of code. We also identified a similar issue in the `verifyClaims()` routine on the `receivers` (lines 81 and 93 in the `Verifier_Receiver` contract).

Recommendation Revise the two above `verifyClaims()` routines to avoid the contradictory requirements.

Status The issue has been fixed by this commit: `9ee8f5a692e666160b4305d152ff7223859adfa4`.

3.5 Improved Sanity Checks of Function Arguments

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Pot`, `LuckManager`, `FundManager`, `Producer`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

Throughout the current codebase, we observe a number of functions can be benefited from performing more rigorous sanity checks on provided arguments. In the following, we show several representative cases.

Case I: In the following, we show the code implementation of `updateSupportedTokenList()` that allows the governance to update the set of supported tokens. This routine can be improved by applying a check to verify the two arrays have the same length. A similar observation is also applicable to the `initialize()` routine of the same contract.

```
    function updateSupportedTokenList(
        bytes32[] memory _tokens,
        address[] memory _tokenContracts
    ) public onlyGovernanceEnforcer returns (bool) {
        for (uint256 i = 0; i < _tokens.length; i++) {
            tokens[_tokens[i]] = _tokenContracts[i];
        }
        tokenList = _tokens;
        return true;
    }
```

Listing 3.7: Pot.sol

Case II: The second case is the `initialize()` routine in the `LuckManager` contract. The particular routine can also be improved by ensuring the same length of given arguments, i.e., `require(_roles.length = _luckPerRoles.length, "LuckManager: Invalid Input")`.

```
    function initialize(
        address _governanceEnforcerProxy,
        address _pot,
        bytes32[] memory _roles,
        uint256[][] memory _luckPerRoles
    ) public initializer {
        for (uint256 i = 0; i < _luckPerRoles.length; i++) {
            require(_luckPerRoles[i].length == 7, "LuckManager: Invalid Input");
            luckByRoles[_roles[i]] = LuckPerRole(
                _luckPerRoles[i][0],
                _luckPerRoles[i][1],
                _luckPerRoles[i][2],
                _luckPerRoles[i][3],
                _luckPerRoles[i][4],
                _luckPerRoles[i][5]
            );
            luckByRoles[_roles[i]]
                .luckLimit[_luckPerRoles[i][3]] = _luckPerRoles[i][6];
        }
        GovernanceEnforcerProxy = _governanceEnforcerProxy;
        pot = Pot(_pot);
    }
```

Listing 3.8: LuckManager.sol

Case III: The third case is the `updateFundInflation()` routine in the `FundManager` contract. We can naturally enforce the following requirement as the updated `_epochOfUpdate` needs to occur before the fund expires: `require(_epochOfUpdate <= endEpoch, "FundManager: Invalid Input")`.

Case IV: One last case is the `addProducer()` routine in the `Producer` contract. We can ensure the new `baseChainProducer` should not be `address(0)`. Note that the current code logic allows everyone to arbitrarily set `address(0)` as the producer.

```
require(_epochOfUpdate <= endEpoch, "addProducer: Invalid Input").
```

```
function addProducer(address _producer, bytes memory _sig) public {
    bytes32 sigPayload = createPayloadToSig(_producer);
    address baseChainProducer = recoverSigner(sigPayload, _sig);
    bytes memory baseChainProducerAsBytes = abi.encodePacked(
        baseChainProducer
    );
    producerData[keccak256(baseChainProducerAsBytes)] = _producer;
}
```

Listing 3.9: Producer.sol

Recommendation Apply additional sanity checks as suggested above to filter possibly bad inputs.

Status The issue has been fixed by this commit: `710df8d6a0ecdedcc9264416ba3b51276dedeeea`. Note the case III does not need to be addressed as the `governance` may decide to fund after the initial funding ends, and the `governance` might want to change the inflation before it funds the `pot` again.

3.6 Possible Pot Overdraw From FundManager

- ID: PVE-006
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `FundManager`
- Category: Business Logics [9]
- CWE subcategory: CWE-837 [5]

Description

The `pot` contract plays an important role in the fund distribution process. Specifically, it collects the rewards from `FundManager` and re-distributes to other actors, i.e., `receivers` and `producers`.

The Marlin Protocol has developed an incentive mechanism that allows the governance to dynamically apply certain inflation changes. When the rewards are collected by `pot`, the `pot` calls an internal helper `handleInflationChange()` if the next inflation update epoch is already due. However, this helper routine erroneously takes `_currentEpoch`, instead of `fund.nextInflationUpdateEpoch`, for the calculation of `fundToWithdraw` (line 346), hence leading to overdraw from `FundManager`.

```
341     function handleInflationChange(address _pot, Fund memory fund, uint256 _currentEpoch
        )
342         private
```

```
343         returns (uint256 fundToWithdrawBeforeInflationChange)
344     {
345         uint256 fundToWithdraw = fund.inflationPerEpoch.mul(
346             _currentEpoch.sub(fund.lastDrawnEpoch)
347         );
348         require(fund.inflationEpochLogIndex < 5, "Draw  before performing more
                operations");
349         fund.inflationEpochLog[fund.inflationEpochLogIndex] = fund.
                nextInflationUpdateEpoch;
350         fund.inflationLog[fund.inflationEpochLogIndex-1] = fund
351             .inflationPerEpoch;
352         fund.inflationEpochLogIndex++;
353         fund.inflationPerEpoch = fund.nextInflation;
354         fund.lastDrawnEpoch = fund.nextInflationUpdateEpoch;
355         fund.nextInflation = 0;
356         fund.nextInflationUpdateEpoch = MAX_INT;
357         funds[_pot] = fund;
358         return fundToWithdraw;
359     }
```

Listing 3.10: FundManager.sol

Recommendation The fix is rather straightforward as we can just provide `fund.nextInflationUpdateEpoch` (line 346) as follows.

```
341     function handleInflationChange(address _pot, Fund memory fund, uint256 _currentEpoch
            )
342         private
343         returns (uint256 fundToWithdrawBeforeInflationChange)
344     {
345         uint256 fundToWithdraw = fund.inflationPerEpoch.mul(
346             _fund.nextInflationUpdateEpoch.sub(fund.lastDrawnEpoch)
347         );
348         require(fund.inflationEpochLogIndex < 5, "Draw  before performing more
                operations");
349         fund.inflationEpochLog[fund.inflationEpochLogIndex] = fund.
                nextInflationUpdateEpoch;
350         fund.inflationLog[fund.inflationEpochLogIndex-1] = fund
351             .inflationPerEpoch;
352         fund.inflationEpochLogIndex++;
353         fund.inflationPerEpoch = fund.nextInflation;
354         fund.lastDrawnEpoch = fund.nextInflationUpdateEpoch;
355         fund.nextInflation = 0;
356         fund.nextInflationUpdateEpoch = MAX_INT;
357         funds[_pot] = fund;
358         return fundToWithdraw;
359     }
```

Listing 3.11: FundManager.sol

Status The issue has been fixed by this commit: `9ee8f5a692e666160b4305d152ff7223859adfa4`.

3.7 Back-Running of draw() For Denial-of-Service

- ID: PVE-007
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: FundManager
- Category: Business Logics [9]
- CWE subcategory: CWE-837 [5]

Description

As discussed in Section 3.6, the pot contract is responsible for collecting the rewards from FundManager and then re-distributing them to other actors, including receivers and producers. The reward-collection logic is implemented in the draw() routine.

At the core, the draw() routine is tasked to calculate the withdraw amount that is credited to a pot and reduce the amount from the internal balance (fundBalance – line 330). However, we point out that current implementation does not immediately withdraw the amount from FundManager. Instead, it adjusts the allowance the pot is permitted to collect. This immediately leads to a concern in that if a malicious actor can launch a back-running attack to call other functionalities in the same contract, it could mess up internal states, leading to a denial-of-service attack on FundManager.

```
326         require(
327             fundBalance >= withdrawalAmount,
328             "Balance with fund not sufficient"
329         );
330         fundBalance = fundBalance.sub(withdrawalAmount);
331         uint256 approvedTokens = LINProxy.allowance(address(this), _pot);
332         withdrawalAmount = withdrawalAmount.add(approvedTokens);
333         require(
334             LINProxy.approve(_pot, withdrawalAmount),
335             "Fund not allocated to pot"
336         );
337         emit FundDrawn(_pot, withdrawalAmount, fundBalance);
```

Listing 3.12: FundManager.sol

This is indeed the case. Specifically, there is a candidate function updateLINAllocation() that can be exploited to restore the fund balance to the original amount right before the pot reward reduction. Moreover, it unintentionally increases an internal state, i.e., unallocatedBalance. This states records the unallocated balance that is available for other pots to collect, thus possibly retrieving more funds than normally entitled from FundManager.

```
    // Note: If this function execute successfully when new fund
    // wasn't allocated to rewards, then something is not right
    // The update should never result in lower Balance than the fund balance
    //todo: Is it a security risk to keep this open, just in case, there is a
```

```
    // leak in the way fundbalance or unallocated balance is calculated.
    function updateLINAllocation() public {
        uint256 _fundBalance = LINProxy.balanceOf(address(this));
        require(
            _fundBalance != fundBalance,
            "Fund Balance is as per the current balance"
        );
        emit FundBalanceUpdated(fundBalance, _fundBalance);
        unallocatedBalance = unallocatedBalance.add(_fundBalance).sub(
            fundBalance
        );
        fundBalance = _fundBalance;
    }
```

Listing 3.13: FundManager.sol

Recommendation Instead of only increasing the reward allowance for the `pot`, actually transfer the amount out to `pot`.

Status The issue has been fixed by this commit: `9ee8f5a692e666160b4305d152ff7223859adfa4`.

3.8 Incompatibility With Deflationary Tokens in Pot

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Pot`
- Category: Business Logics [9]
- CWE subcategory: CWE-708 [4]

Description

In Marlin Protocol, the `pot` contract operates as the reward distribution center from `FundManager` to other actors. There are two basic operations, i.e., `addToPot()` and `claimFeeReward()`. The first one adds funds into the `pot` while the second one retrieves funds from the `pot`.

Naturally, the above two functions, i.e., `addToPot()` and `claimFeeReward()`, are involved in transferring reward assets into (or out of) the `pot`. Using the `addToPot()` function as an example, it needs to transfer rewards from `FundManager` to `pot` (lines 225−229). When transferring standard ERC20 tokens, these asset-transferring routines work as expected: namely the account's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts (lines 208 − 213).

```
198    // Note: These tokens should be approved by governance else can be attacked
199    function addToPot(
200        uint256[] memory _epochs,
```

```
        address _source,
        bytes32 _token,
        uint256[] memory _values
    ) public returns (bool) {
        require(_epochs.length == _values.length, "Pot: Invalid inputs");
        uint256 totalValue;
        for (uint256 i = 0; i < _epochs.length; i++) {
            uint256 updatedPotPerEpoch = potByEpoch[_epochs[i]].value[_token]
                .add(_values[i]);
            potByEpoch[_epochs[i]].value[_token] = updatedPotPerEpoch;
            potByEpoch[_epochs[i]].currentValue[_token] = potByEpoch[_epochs[i]]
                .currentValue[_token]
                .add(_values[i]);
            emit PotFunded(
                msg.sender,
                _epochs[i],
                _token,
                _source,
                _values[i],
                updatedPotPerEpoch
            );
            totalValue = totalValue.add(_values[i]);
        }
        require(
            IERC20(tokens[_token]).transferFrom(
                _source,
                address(this),
                totalValue
            ),
            "Pot: Couldn't add to pot"
        );
        return true;
    }
```

Listing 3.14: Pot.sol

However, in the cases of deflationary tokens, as shown in the above code snippet, the input amount may not be equal to the received amount due to the charged (and burned) transaction fee. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `addToPot()` and `claimFeeReward()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts in the cases of deflationary tokens. Apparently, these balance inconsistencies are damaging to accurate pool management and affects protocol-wide operation and maintenance. (And keep in mind that `USDT` may become deflationary if the control switch in its token contract is turned on.)

One mitigation is to query the asset change right before and after the asset-transferring routines. In other words, instead of automatically assuming the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()`/`transferFrom()` is expected and aligned well with the intended operation. Though these additional checks cost additional gas usage, we feel that they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary. Another mitigation is to regulate the set of ERC20 tokens that are permitted into the protocol. With the nature of choosing possible assets with the governance, it is possible to effectively regulate the set of assets allowed into the protocol.

Recommendation Regulate the set of asset tokens supported in Marlin Protocol and, if there is a need to support deflationary tokens, add necessary mitigation mechanisms to keep track of accurate balances.

Status This issue has been confirmed. As there is no comprehensive solution yet, the team decides no change for the time being, but will regulate the set of supported tokens via governance.

3.9 Suggested Constants For Unchanged Variables

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Producer`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [3]

Description

Ethereum smart contracts are typically immutable by default. Once they are created, there is no way to alter them, effectively acting as an unbreakable contract among participants. In the meantime, there are several scenarios where there is a need to upgrade the contracts, either to add new functionalities or mitigate potential bugs.

The upgradeability support comes with a few caveats. One important caveat is related to the initialization of new (logic) contracts that are just deployed to replace old (logic) contracts. Due to the inherent requirement of any proxy-based upgradeability system, no constructors can be used in upgradeable contracts. This means we need to change the constructor of a new contract into a regular function (typically named `initialize()`) that basically executes all the setup logic.

The Marlin Protocol has extensively adopted the above approach. When examining all these `initialize()` routines, we notice the routine in the `Product` contract has the following initialization logic.

```
contract Producer is Initializable {
    bytes id;
    bytes byteVersion;
    bytes id_extended;
```

```
    mapping(bytes32 => address) producerData;

    function initialize() public initializer {
        id = "\x19";
        byteVersion = "03";
        id_extended = "Ethereum Signed Message:\n";
    }
    ...
}
```

Listing 3.15: Producer.sol

Apparently, its initialization essentially assigned constants to internal variables. Since these variables are never changed, it is strongly suggested to re-define them as constants for reduced gas cost.

Recommendation Define those variables as constants for reduced gas cost.

Status The issue has been confirmed. But since the protocol takes a proxy-based approach, declaring variables in the initializer was intentional so that data is stored in the proxy contract rather than the logic contract. With that, as the data needs to be stored in the proxy contract, we cannot directly write the variables as constants.

3.10 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.0` instead of specifying a range, e.g., `pragma solidity >=0.4.21 <0.7.0`.

In addition, there is a known compiler issue that in all 0.5.x solidity prior to `Solidity 0.5.17`. Specifically, a private function can be overridden in a derived contract by a private function of the same name and types. Fortunately, there is no overriding issue in this code, but we still recommend using `Solidity 0.5.17` or above.

Moreover, we strongly suggest not to use experimental Solidity features or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet.

4 Conclusion

In this audit, we thoroughly analyzed the Marlin Protocol design and implementation. The protocol presents a high-performance programmable network infrastructure which could benefit all kinds of blockchains and DApps. During the audit, we noticed that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [12, 13, 14, 15, 17].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [18] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-708: Incorrect Ownership Assignment. https://cwe.mitre.org/data/definitions/708.html.

[5] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[7] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[8] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[9] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[10] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[11] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[12] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[13] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[14] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[15] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

[16] PeckShield. PeckShield Inc. https://www.peckshield.com.

[17] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[18] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.